Beautiful Code

Qu'est-ce que du « beau » code ?

Emmanuel Deloget

Présentation donnée le 18/04/2013 à Polytech Marseille

Plan

- Quoi ?
- Pourquoi ?
- Comment ?
- Conclusion
- Des questions ?

Quoi ?

- Qu'est-ce que du beau code ?
  - Est-ce qu'on peut définir clairement ce qu'est du beau code ?
  - Est-ce que c'est utile ?
  - Est-ce que le « beau code » a vraiment un intérêt ?

Pourquoi ?

- Dans les années 50 : premiers pas de l'informatique
  - Machines sommaires
    - Interface sommaire pour écrire du code : cartes perforées, stockage sur bande magnétiques...
    - Systèmes très chers, utilisation très limitée
  - Apparition des premiers langages (Fortran 1954, Algol 1958...)
  - « **Comment écrire du code correct et efficace ?** »
    - En le compilant

Pourquoi ? (suite)

- Dans les années 70
  - Les machines se complexifient : plus de RAM, des systèmes de stockage plus performants, …
    - Les programmes se complexifient aussi
      - On code maintenant des OS dans un langage haut niveau (UNIX, 1972, écrit en C) !
    - Ils deviennent de plus en plus critiques : systèmes boursiers, avionique...
    - Il doivent être pérennes dans le temps
  - « **Comment écrire du code correct, efficace et maintenable ?** »
    - Premiers pas de la programmation objet (Smalltalk, 1971)
    - Code architecturé

Pourquoi ? (suite)

- Dans les années 80/90 : le règne de la micro-informatique
  - Arrivée des micro-ordinateurs personnels (Apple II, Altair...) et professionnels (IBM PC et ses clones)
  - La demande en programmes explose
    - Traitements de texte, tableurs, comptabilité...
  - « **Comment écrire simplement du code correct, efficace et maintenable ?** »
    - Code architecturé
    - Et puis ?

Pourquoi ? (fin)

- Problème : je sais dire si mon code est correct et efficace
  - J'ai des outils pour ça : compilateur, debugger, analyse statique du code...
- Comment est-ce que je sais s'il est simple et maintenable ?
  - Réponse classique : « le beau code est simple et maintenable ».
- Oui, certes. Mais c'est quoi, du beau code ?
  - Réponse classique : « Euh... »

Comment ?

- Que veut dire « beau » ?
  - A moins de se lancer dans un grand débat philosophique, ça va être difficile de trancher.
    - Hippias Majeur, de Platon (sur Wikisource)
  - Psychologie : théorie de la fluidité du traitement (*processing fluency theory*)
    - Plus on maîtrise un sujet, plus on y voit des valeurs esthétiques

Comment ? (suite)

- Chez les mathématiciens : une démonstration dite « élégante » (ou belle ; source : Wikipedia)
  - Utilise peu de résultats préalables
  - Est exceptionnellement courte
  - Établit le résultat d'une manière surprenante
  - Fait appel à une méthode qui peut être généralisée
- Belle formule mathématique
  - $e^{i\pi} + 1 = 0$
- Pas tout à fait le même but – la visée des mathématiciens est purement esthétique.
  - Les mathématiciens n'ont pas à assurer la maintenance de leurs démonstrations...

Comment ? (fin)

- On va se contenter de « qui a un ensemble de caractéristiques qui nous semblent désirables »
  - Validité
  - Simplicité
  - Lisibilité
  - Expressivité
  - Efficacité
- Méthode de travail
  - On prend un critère
  - On le définit
  - Si possible, on le teste
  - On propose une conclusion

Critère 1 : validité du code

- Définition : un code est valide s'il est exempt d'erreur
  - **Erreur lexicale** : faute d'orthographe empêchant la compréhension du code
  - **Erreur syntaxique** : faute de grammaire empêchant la compréhension du code
  - **Erreur sémantique** : le code fait quelque chose qui n'a pas de sens
- Les erreurs lexicales et syntaxiques sont découvertes automatiquement par le compilateur
  - Il est souvent assez facile de les corriger
  - Certains langages ne nous simplifient pas la tâche : C++ avec ses templates est un bon exemple.
- Les erreurs sémantiques ne le sont pas – dans la grande majorité des cas.

Critère 1 : validité du code

- Exemple (C)
  - Le code compile (avec warning sur les compilateurs récents)
  - A l'exécution, il plante...

```
int main(void)
{
	int valeur = 10 ;
	printf("on affiche une chaîne :%s\n", valeur) ;
}
```

Critère 1 : validité du code

- Si un programme contient une erreur, alors ce programme ne peut être complètement analysé
  - Certaines erreurs sont très facilement visibles, d'autres non
- Dans l'absolu, on ne peut pas juger ce qu'on ne peut pas analyser
  - Si le code ne compile pas, il est inutile de chercher à définir ses qualités.
  - Le code invalide ne peut pas être pas « beau » – non pas qu'il ne soit pas « beau », mais parce qu'on ne peut pas juger de ses qualités
- Dans la pratique : 99,9 % des programmes sont buggés (et contiennent donc au moins une erreur sémantique)
  - On laisse donc de côté les erreurs sémantiques qui n'ont pas encore été découvertes.
  - Ca nous permet de juger le code

Critère 1 : validité du code

- Conclusion 1a : **si le code ne compile pas ou présente une erreur sémantique grossière, ça ne sert à rien de le juger.**
- Conclusion 1b : **la présence ou non d'erreurs sémantiques non découvertes ne peut pas nous empêcher de juger le code**
- Au final : Ce critère est d'une étonnante inutilité.

Critère 2 : simplicité

- Le code est simple s'il fait les choses simplement
  - *x = y * 2* ou *x = y << 1* ?
- Faire les choses simplement requiert une bonne connaissances des outils à notre disposition
  - Connaître le langage et ses constructions
  - Connaître les librairies utilisées
- **La simplicité est contextuelle**

Critère 2 : simplicité

- Exemples de code
  - Le code (en C) calcule la longueur maximale d'une chaîne de caractère afin de l'allouer et de la retourner à l'appelant.
  - Code complexe : les commentaires sont obligatoires pour comprendre ce que le code fait, et pourquoi il le fait.
  - Code simple : un peu plus court.

Critère 2 : simplicité

- Faire simple, ce n'est pas nécessairement faire court
  - Il peut être utile parfois de faire plus long, de manière à mieux expliciter un algorithme lui-même complexe
- Rasoir d'Ockham : « *Les multiples ne doivent pas être utilisés sans nécessité.* »
  - Ce qui est simple dans le principe doit être fait simplement.
  - C'est encore plus vrai quand on s'attaque à une idée complexe.
- Le but de la simplicité est d'aider à la compréhension du code
- Trop simple, ce n'est pas forcément bon : le code devient naïf (souvent signe d'une méconnaissance des outils utilisés).

Critère 2 : simplicité

- Conclusions :
  - Un code qui n'est pas simple ne peut pas être beau, parce que sa complexité nuit à sa compréhension.
  - Un code qui est trop simple n'est pas beau, il est naïf.
- **Le chemin qui mène à la « beauté » passe par une juste simplicité**
  - Connaissance des outils
  - Prise en compte du contexte
- **Cette juste simplicité naît de la théorie et de l'expérience**

Critère 3 : lisibilité

- Un code est lisible s'il est déchiffrable
  - Utilisation d'un style de programmation qui favorise la lisibilité
    - Espaces, placement des parenthèses ou des ouvertures de bloc…
    - Utilisation de constructions connues pour limiter le besoin de déchiffrement
    - Simplifier les expressions pour qu'elles ne soient pas trop longues
    - ...

Critère 3 : lisibilité

- Exemples :
  - De l'utilisation des caractère d'espacement
    - Quel est le code embrouillé, quel est le code simple ?
  - De l'utilisation des constructions complexes ou simples
    - Quelle construction sera plus facile à utiliser ?
  - Quid des langages ésotériques ?

Critère 3 : lisibilité

- Un code embrouillé ne peut pas être beau – il est juste embrouillé
- Simplicité et lisibilité sont quelquefois liés : un code trop complexe sera plus difficilement déchiffrable.
- **Un code beau est nécessairement un code lisible**.
  - Mais ce n'est pas parce qu'il est lisible qu'un code est beau.

Critère 4 : expressivité

- Un code expressif est un code qui dit ce qu'il fait malgré sa brièveté
  - Le besoin en commentaire est réduit
  - Le code reste lisible et simple
- Un code expressif a du sens
  - Un programme est une abstraction.
  - Toute abstraction doit avoir un sens.
  - Ce sens doit se retrouver dans les termes utilisés.

Critère 4 : expressivité

- Exemples :
  - Multiplication de matrices
  - Nommage
  - Sémantique
- L'expressivité est fortement liée aux possibilités du langage
  - Certains langages sont plus expressifs que d'autres
    - C'est pareil pour les langues parlées.
  - On doit chercher à atteindre l'expressivité maximale que le langage nous autorise.

Critère 4 : expressivité

- Simplicité, lisibilité et expressivité sont liés
  - Un code expressif sera simple et lisible
  - Un code qui ne l'est pas sera plus complexe ou sera difficile à déchiffrer
- Un code peu ou pas expressif rend plus complexe la maintenance d'un programme.
- **Pour être beau, un code doit donc être expressif**.

Critère 5 : efficacité

- Un code est efficace s'il utilise un minimum de ressources
  - Temps CPU
    - Algorithme optimal ou (au moins) acceptable en terme de complexité
    - Implémentation correcte de l'algorithme
    - Optimisations supplémentaires
  - Mémoire
  - ...

Critère 5 : efficacité

- Exemples
  - monkey_sort()
  - rsqrt()
- « Le code optimisé n'est pas lisible ! »
  - Vrai et faux
    - Donald Knuth : « *We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.* »
    - Mais la suite de la citation : « *Yet we should not pass up our opportunities in that critical 3%.* »
  - Rappel : la simplicité est contextuelle et requiert quelquefois des connaissances particulières

Critère 5 : efficacité

- Un code trop lent, utilisant trop de ressources ne peut être considéré comme beau. C'est un code gourmand.
- Un code trop (mal?) optimisé ne peut pas être beau – pour d'autres raisons
  - Code illisible, inexpressif...
- **Un code beau est efficace, mais son efficacité n'est pas obtenue aux dépens d'autres qualités.**

Conclusion (début)

- Pour être beau,
  - un code doit pouvoir être jugé (et doit donc être valide).
  - Il doit être simple, lisible et expressif
  - Il doit être efficace, sans que cela ne nuise à sa simplicité, sa lisibilité et son expressivité
- Au final, on obtient un code qui est plaisant à lire, facile à comprendre, et stimulant pour l'intellect.
  - C'est tout ?

Conclusion (suite)

- Du fait de ces qualités, le code hérite d'autres qualités intéressantes :
  - Il est plus facile à maintenir et à faire évoluer
    - Il est plus facile de trouver un contresens dans un programme qui exprime clairement son sens : les bugs sont donc plus facilement trouvés et corrigés.
  - Intégrer une personne nouvelle dans l'équipe nécessite une formation moins importante
- **Ce sont ces qualités qui sont intéressantes, parce qu'elles permettent de réduire le coût de développement et de maintenance**.

Conclusion (suite)

- Comment faire pour être sûr d'écrire du « beau » code ?
  - Hélas, c'est une autre histoire...
  - Nécessite des connaissances théoriques (algorithmique, systèmes, langage de programmation...) et pratiques (expérience)
  - **Nécessite surtout d'avoir un regard critique sur son propre code et sur celui des autres.**
    - Analyse des défauts
    - Comprendre pourquoi « ce n'est pas bien »

Conclusion (fin)

- Est-ce que ça va me servir plus tard ?
  - **OUI !**
  - Si vous programmez beaucoup,
    - Pour gagner du temps
    - Et surtout pour ne pas en perdre
  - Si vous gérez des programmeurs
    - Pour évaluer la qualité de leur travail
    - Pour les guider
  - Le but, au final, est de réduire les coûts de développement d'un produit ou d'une application.
    - Et si vous êtes votre propre patron, c'est de vos sous dont on parle :)

Des questions ?

Programmez !